DAW Cliente - Corolario

Para realizar una aplicación web se debe tener:

Una api del lado servidor

Una api del lado cliente

Cada una de las apis debería se creada a parir de clases.

Lado servidor:

rutas.js:

```
const express = require('express');
const router = express.Router();
const UserController = require('./controllers/UserController');

const userController = new UserController();

router.get('/users', userController.getAllUsers.bind(userController));
router.get('/users/:id', userController.getUserById.bind(userController));
router.post('/users', userController.createUser.bind(userController));
router.put('/users/:id', userController.updateUser.bind(userController));
router.delete('/users/:id', userController.deleteUser.bind(userController));

module.exports = router;
```

claseControladora.js

```
class UserController {
    async getAllUsers(req, res) {
      // Lógica para obtener todos los usuarios
      // Puede acceder a la base de datos, realizar consultas, etc.
      const users = await User.find();
      res.json(users);
    }

    async getUserById(req, res) {
      const userId = req.params.id;
      // Lógica para obtener un usuario por ID
      const user = await User.findById(userId);
      if (!user) {
        return res.status(404).json({ error: 'Usuario no encontrado' });
```

```
      }
      res.json(user);
    }

    async createUser(req, res) {
      const { name, email, password } = req.body;
      // Lógica para crear un nuevo usuario
      const newUser = new User({ name, email, password });
      await newUser.save();
      res.status(201).json(newUser);
    }

    async updateUser(req, res) {
      const userId = req.params.id;
      const { name, email, password } = req.body;
      // Lógica para actualizar un usuario por ID
      const updatedUser = await User.findByIdAndUpdate(userId, { name, email, password
}, { new: true });
      if (!updatedUser) {
        return res.status(404).json({ error: 'Usuario no encontrado' });
      }
      res.json(updatedUser);
    }

    async deleteUser(req, res) {
      const userId = req.params.id;
      // Lógica para eliminar un usuario por ID
      const deletedUser = await User.findByIdAndDelete(userId);
      if (!deletedUser) {
        return res.status(404).json({ error: 'Usuario no encontrado' });
      }
      res.json(deletedUser);
    }
  }

  module.exports = UserController;
```

Lado Cliente:

apiCliente.js

```
class ApiClient {
    constructor(baseURL) {
      this.baseURL = baseURL;
    }

    async fetchData(endpoint) {
      try {
        const response = await fetch(`${this.baseURL}/${endpoint}`);
```

```
      const data = await response.json();
      return data;
    } catch (error) {
      console.error('Error fetching data:', error);
      throw error;
    }
  }

  async postData(endpoint, body) {
    try {
      const response = await fetch(`${this.baseURL}/${endpoint}`, {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify(body),
      });
      const data = await response.json();
      return data;
    } catch (error) {
      console.error('Error posting data:', error);
      throw error;
    }
  }

  async putData(endpoint, body) {
    try {
      const response = await fetch(`${this.baseURL}/${endpoint}`, {
        method: 'PUT',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify(body),
      });
      const data = await response.json();
      return data;
    } catch (error) {
      console.error('Error updating data:', error);
      throw error;
    }
  }

  async patchData(endpoint, body) {
    try {
      const response = await fetch(`${this.baseURL}/${endpoint}`, {
        method: 'PATCH',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify(body),
      });
      const data = await response.json();
```

```
      return data;
    } catch (error) {
      console.error('Error patching data:', error);
      throw error;
    }
  }

  async deleteData(endpoint) {
    try {
      const response = await fetch(`${this.baseURL}/${endpoint}`, {
        method: 'DELETE',
      });
      const data = await response.json();
      return data;
    } catch (error) {
      console.error('Error deleting data:', error);
      throw error;
    }
  }
}

// Ejemplo de uso
const api = new ApiClient('https://tu-api-express.com');

// Utilizar los métodos según sea necesario
api.fetchData('ruta-api')
  .then(data => console.log('Datos recibidos:', data))
  .catch(error => console.error('Error al obtener datos:', error));

const dataToSend = { key: 'value' };

api.postData('ruta-api', dataToSend)
  .then(response => console.log('Respuesta del servidor (POST):', response))
  .catch(error => console.error('Error al enviar datos (POST):', error));

api.putData('ruta-api/1', dataToSend)
  .then(response => console.log('Respuesta del servidor (PUT):', response))
  .catch(error => console.error('Error al enviar datos (PUT):', error));

api.patchData('ruta-api/1', { updatedKey: 'updatedValue' })
  .then(response => console.log('Respuesta del servidor (PATCH):', response))
  .catch(error => console.error('Error al enviar datos (PATCH):', error));

api.deleteData('ruta-api/1')
  .then(response => console.log('Respuesta del servidor (DELETE):', response))
  .catch(error => console.error('Error al eliminar datos:', error));
```

Mysql y async/await (necesita módulo mysql2 en express)

Se puede utilizar async/await para hacer consultas asícronas a mysql (en node express requiere el módulo mysql2)

`mysql2` proporciona una versión basada en promesas de sus métodos, lo que facilita el uso de **`async/await`**. Es importante gestionar los errores correctamente y cerrar adecuadamente las conexiones o el pool de conexiones cuando sea necesario.

ejemplo, la función **`getAllUsers`** está marcada como **`async`**, y utilizamos **`await pool.execute(query)`** para ejecutar la consulta de forma asíncrona y esperar a que se complete antes de continuar con el código.

En **`mysql2`**, la función **`execute`** no retorna una Promesa directamente, pero puedes envolverla en una Promesa utilizando **`util.promisify`**. Aquí te dejo un ejemplo:

Primero, necesitarás importar el módulo **`util`** de Node.js para usar la función **`promisify`**:

```
const util = require('util');
const mysql = require('mysql2');

const pool = mysql.createPool({
  host: 'localhost',
  user: 'tu_usuario',
  password: 'tu_contraseña',
  database: 'tu_base_de_datos',
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0,
});

// Utilizar promisify para envolver la función execute
const executeAsync = util.promisify(pool.execute).bind(pool);

class UserController {
  async getAllUsers(req, res) {
    try {
      // Lógica para obtener todos los usuarios
      const query = 'SELECT * FROM users'; // Asegúrate de que tu tabla de usuarios se
llama 'users'

      const [rows, fields] = await executeAsync(query);

      res.json(rows);
    } catch (error) {
      console.error('Error al obtener usuarios:', error);
      res.status(500).json({ error: 'Error interno del servidor' });
    }
  }

  // ... otros métodos del controlador
}
module.exports = UserController;
```

En este ejemplo, hemos envuelto la función `execute` en una versión promisificada llamada `executeAsync`. Ahora, puedes utilizar `await executeAsync(query)` para ejecutar la consulta de forma asíncrona.

Esta es una manera de trabajar con `async/await` y `mysql2` cuando la función que necesitas usar no retorna directamente una Promesa.

Lo mismo sin necesidad de pool en la base de datos:

```
const util = require('util');
const mysql = require('mysql2');

// Configuración de la conexión a la base de datos
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'tu_usuario',
  password: 'tu_contraseña',
  database: 'tu_base_de_datos',
});

// Utilizar promisify para envolver la función query
const queryAsync = util.promisify(connection.query).bind(connection);

class UserController {
  async getAllUsers(req, res) {
    try {
      // Lógica para obtener todos los usuarios
      const query = 'SELECT * FROM users'; // Asegúrate de que tu tabla de usuarios se
llama 'users'

      const [rows, fields] = await queryAsync(query);

      res.json(rows);
    } catch (error) {
      console.error('Error al obtener usuarios:', error);
      res.status(500).json({ error: 'Error interno del servidor' });
    }
  }

  // ... otros métodos del controlador
}

module.exports = UserController;
```

Cerrar las conexiones a la base de datos:

Para cerrar la conexión a la base de datos en **`mysql2`**, puedes utilizar el método **`end`**. Aquí tienes un ejemplo de cómo puedes hacerlo:

```
const mysql = require('mysql2');

// Configuración de la conexión a la base de datos
const connection = mysql.createConnection({
```

```
  host: 'localhost',
  user: 'tu_usuario',
  password: 'tu_contraseña',
  database: 'tu_base_de_datos',
});

// Lógica para obtener todos los usuarios
async function getAllUsers() {
  try {
    const query = 'SELECT * FROM users'; // Asegúrate de que tu tabla de usuarios se
llama 'users'

    const [rows, fields] = await connection.promise().query(query);

    console.log(rows);

    // Cerrar la conexión después de usarla
    connection.end();
  } catch (error) {
    console.error('Error al obtener usuarios:', error);
  }
}

// Llamada a la función
getAllUsers();
```

En este ejemplo, después de realizar la consulta y procesar los resultados, he agregado la línea **`connection.end()`** para cerrar la conexión.

Es importante destacar que debes cerrar la conexión después de haber realizado todas las operaciones necesarias en tu aplicación. Si tienes múltiples consultas o interacciones con la base de datos, es recomendable cerrar la conexión una vez que hayas terminado con todas las operaciones relacionadas con la base de datos.

Ten en cuenta que si estás utilizando un enfoque de conexión por solicitud (creando y cerrando conexiones para cada solicitud), esta práctica es más común. Sin embargo, si estás utilizando un pool de conexiones, generalmente no necesitas cerrar la conexión manualmente, ya que el pool maneja la administración de conexiones por ti.

Uso de pool de conexión a MySQL (conectar y liberar)

db.js

```
const mysql = require('mysql2');  // se instala con: npm i mysql
const db = mysql.createPool({
  //propiedad host donde se ubica la base de datos Mysql
  host: '0.0.0.0',
  //port: 3307,     // se puede camiar el puerto de mysql

  user: 'ejemplouser',
  //contraseña del usuario en mysql
  password: '123456',
  //nombre del esquema al que nos vamos a conectar
  database: 'ejemplodb',
```

```
  waitForConnections: true,
  connectionLimit: 10,
  queueLimit: 0,
});
module.exports=db;
```

ciudadesController.js

```
const db = require('../databases/db');

const getCiudades= (req, res) => {  //http://localhost:3000/ciudades
    // Consulta a la base de datos
    // Obtener una conexión del pool
    db.getConnection((err, connection) => {
      if (err) {
        console.error('Error al obtener la conexión del pool:', err);
      } else {
        console.log('Conexión exitosa al pool de conexiones');

        // Realizar operaciones con la conexión
        // Ejemplo: Consulta a la base de datos
        connection.query('SELECT * FROM ciudades', (queryError, results) => {
          if (queryError) {
            console.error('Error en la consulta:', queryError);
          } else {
            console.log('Resultados de la consulta:', results);
            res.json(results)
          }

          // Importante: Liberar la conexión de vuelta al pool cuando hayas terminado
con ella
          connection.release();
        });
      }
    });


  module.exports={
    getCiudades,
    crearCiudad,
    getCiudadById,
    putCiudad,
    patchCiudad,
    actualizarCiudad,
    deleteCiudad,
    getCiudadesByHabitantes,
  };
```